# Basic knowledge AMK PLC programming
# Software: CoDeSys
# Language: Structured text

Version: 2012/36
Part No.: 204019
Translation of the "Original Beschreibung"

## Notes on this document

Name: PDK_204019_Basiswissen_AMK_Steuerungsprogrammierung

Version:

| Version | Change | Letter symbol |
|---|---|---|
| 2012/36 | Example own FB | KoJ |

Previous version: 2011/40

Product version:

| Product | Firmware Version (AMK part-no.) | Hardware Version (AMK part-no.) |
|---|---|---|
| | | |



Reservation: We reserve the right to modify the content of the documentation as well as the delivery options for the product.

Publisher: AMK Arnold Müller GmbH & Co. KG
Gaußstraße 37 - 39
D-73230 Kirchheim/Teck
Germany
Phone: 0049/(0)7021/5005-0
Fax: 0049/(0)7021/5005-176
E-Mail: info@amk-antriebe.de
Managing director: Dr.h.c. Arnold Müller, Eberhard A. Müller, Dr. Günther Vogt
Registration court Stuttgart HRB 231283; HRA 230681

Service: Phone: 0049/(0)7021/5005-190, Fax -193
Office hours:

| | |
|---|---|
| Mon. - Thu. | 7:30 - 12:00am and 1:00 - 4:30pm |
| Fri. | 7:30 - 12:00am and 1:00 - 3:30pm |
| Sat., Sun. and holidays | Please leave your contact data on the answering phone, which is monitored regularly between 9:00am and 17:00pm. Our service will call you back as soon as possible. |

For fast and reliable troubleshooting, you can help us by informing our Customer Service about the following:

- Type plate data for each unit
- Software version
- Device configuration and application
- Type of fault/problem and suspected cause
- Diagnostic messages (error messages)

E-Mail: service@amk-antriebe.de

Internet address: www.amk-antriebe.de

# 1 Display conventions

| Display | Meaning |
|---|---|
|  | This symbol points to parts of the text to which particular attention should be paid. |
| | Parameter names, e.g. "ID2 SERCOS cycle time"<br>Diagnostic message, e.g. "1234 Mains failure" |
| 0x | 0x followed by a hexadecimal number, e.g. 0x500A |
| 'Name'<br>'ID0815 parameter text'<br>'1234 diagnostic message' | e.g.: Call up the function 'delete PLC programme'. |
| **'bold'** | Menu items and buttons in a software or on a control unit, e.g.:<br>Click the **'OK'** button in the **'Options'** menu to call up the 'Delete PLC program' function |
| >Input variable< | A variable that is entered using the operator interface. |
|  | In the examples, the hand symbol shows where to click.<br>RMB: Right mouse button |
| <iValue1> | Variables that need to be created by the user |

# Content

# 2 Introduction

This documentation describes the introduction in programming according to IEC 61131-3 with the programming software CoDeSys and the AMK function libraries. It explains the various modules, variables, data types and the instruction codes of the programming language Structured Text.

Further information on handling CoDeSys can be found in the CoDeSys manual V2.3 documentation that is automatically installed as well.

# 3 CoDeSys

CoDeSys is a PC software for controller programming according to IEC 31131-3. The EN 61131-3 (also IEC 1131 or otherwise 61131) is the only norm for programming languages of programmable-memory controllers that is valid around the world.

AMK is a CoDeSys Automation Alliance Partner. All AMK PLC modules are programmed with CoDeSys.

CoDeSys provides the programming platform according to IEC 61131-3, the basic library and the visualisation editor. AMK provides the PLC hardware, the motion control libraries and preconfigured visualisation objects.

**Programming languages according to EN 61131-3**

IL : Instruction List

LD : Ladder Diagram

FBD : Function Block Diagram

SFC : Sequential Function Chart

ST : Structured Text

AMK applications are preferably programmed in ST. Structured Text is a higher programming language similar to Pascal with additional language tools such as temporal processing and access to process signals.

## 3.1 Functionality of the CoDeSys PC software

- Programming languages according to EN 61131-3
- Task configuration that can process PLC program in various temporal levels
- Controller configuration tool for a process mapping of all I/O addresses
- Library manger, for standard, AMK and own libraries
- General online features
  - Monitoring / writing / forcing of variables
  - Breakpoints / single step / single cycle
  - Online change
  - Downloading / uploading a file
  - Boot project
  - Downloading of source code
- Integrated simulation (standard modules) sampling trace (integrated logic analyser, digital memory oscilloscope (DMO), data logger)
- Recipe and watch manager to monitor variables from various modules
- Network variables
- Integrated visualisation
- Call tree

# 4 POUs

## 4.1 Program blocks PLC_PRG and FPLC_PRG

AMKASYN controllers operate with an asynchronous and a synchronous task. (A task is a temporal procedure unit of an IEC program.)

The asynchronous task is formed by the cyclic program block 'PLC_PRG' and is the main program in the project. It is called up exactly one time per control cycle. The cycle time is not fixed, but results from the length of the program to be processed.

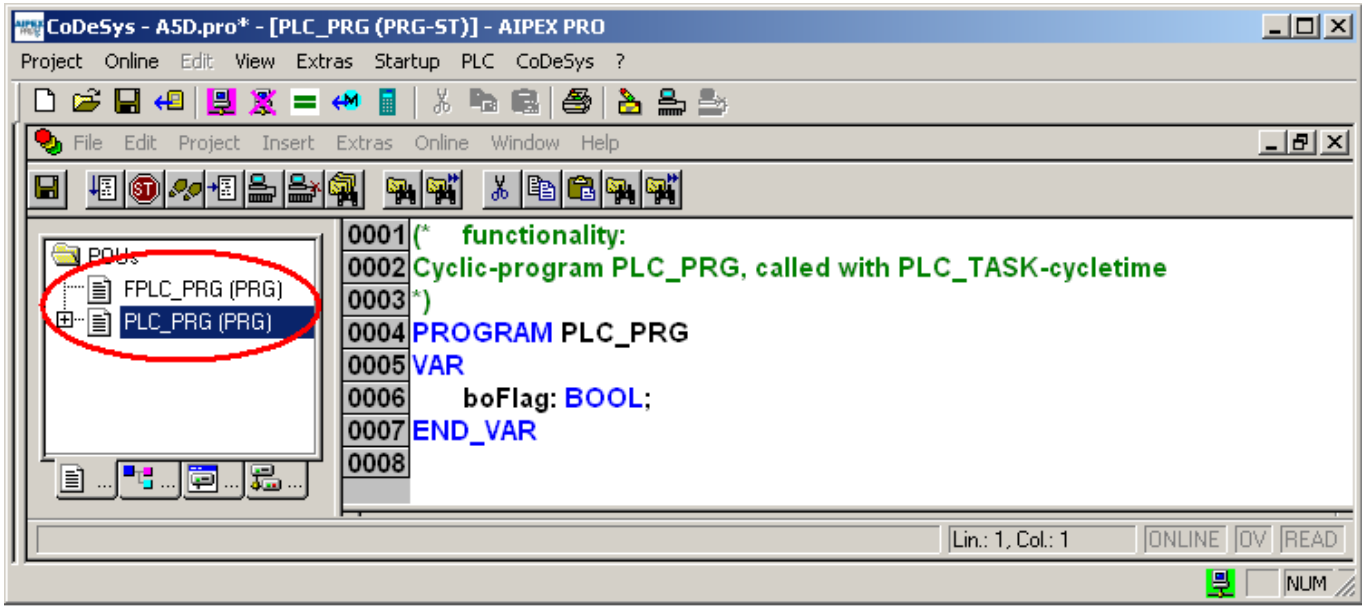
The program block 'PLC_PRG' can call up other modules of the project that are then also processed in the asynchronous task.

The synchronous (real-time) task is formed by the program block 'FPLC_PRG'. The 'FPLC_PRG' is synchronised to the AMKASYN system cycle (PGT), in which the setpoint value channels can be read-in for example.

The 'FPLC_PRG' forms the main program for all cyclic-synchronous module calls in the project. The program block 'FPLC_PRG' can call up other modules of the project that are then also processed in the synchronous task.

The cycle time is fixed and is set by the parameter ID2 'SERCOS cycle time' in the basic device.

Do not delete the modules 'PLC_PRG' and 'FPLC_PRG' and do not rename them either (under the condition that you do not use any task configuration). 'PLC_PRG' is generally the main program in the project.




## 4.2 Function blocks (FB)

Function blocks encapsule a program code with internal variables, i.e. they feature a memory.

The output values ('VAR_OUT', 'VAR_IN_OUT') depend on the input values ('VAR_IN', 'VAR_IN_OUT') and values of the internal variables.
The values of the internal variables are sustained between the call-ups.

An FB is created one time and can be instanced infinitely. An FB is instanced by creating a copy of the FB.
This copy contains a name and is presented to the programming system by variable declaration. (See Example: Instancing function blocks on page 7.)

There are standard function blocks (from AMK or CoDeSys) or user-defined function blocks (created by the user).

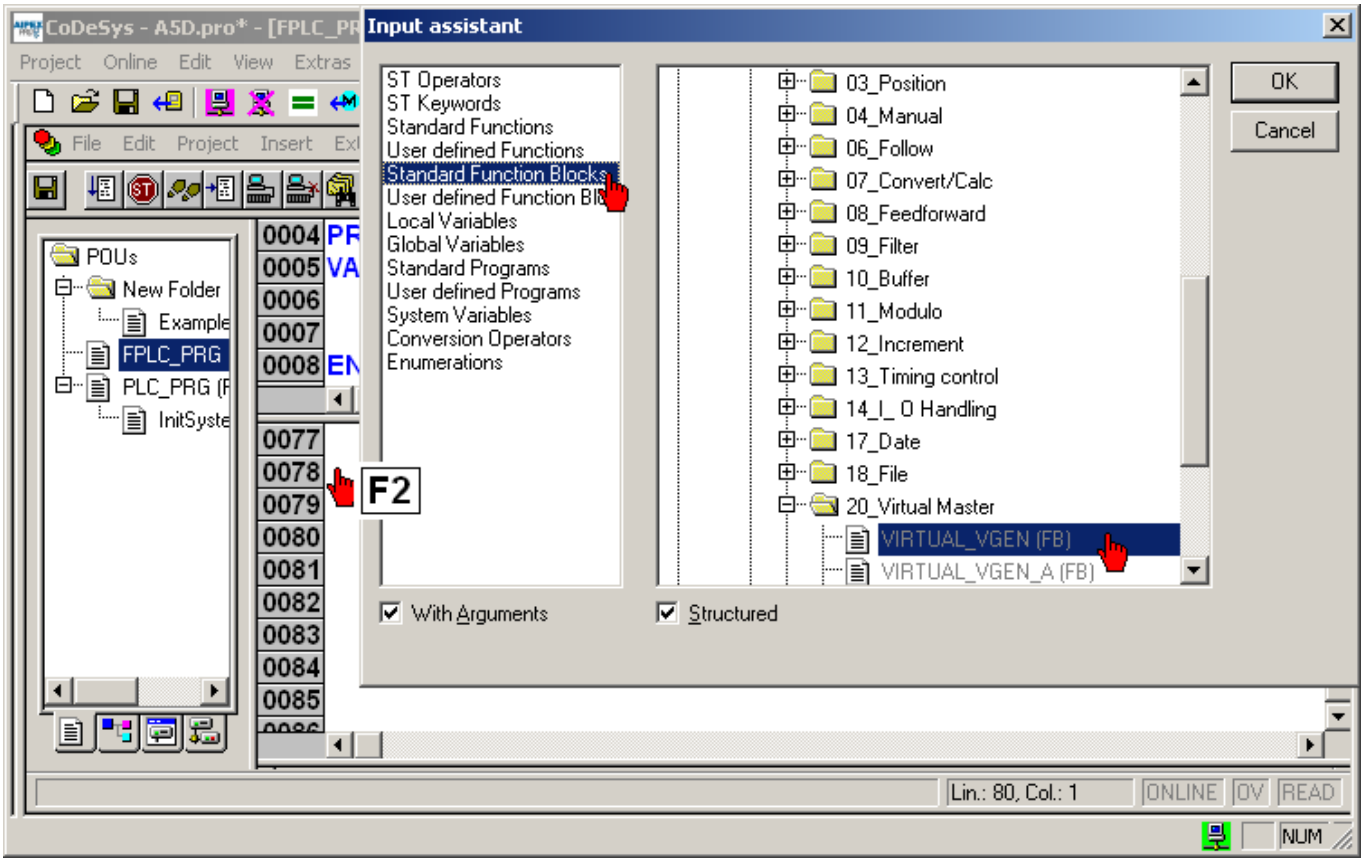
See User-defined function block on page 9.

### 4.2.1 Example: Instancing function blocks

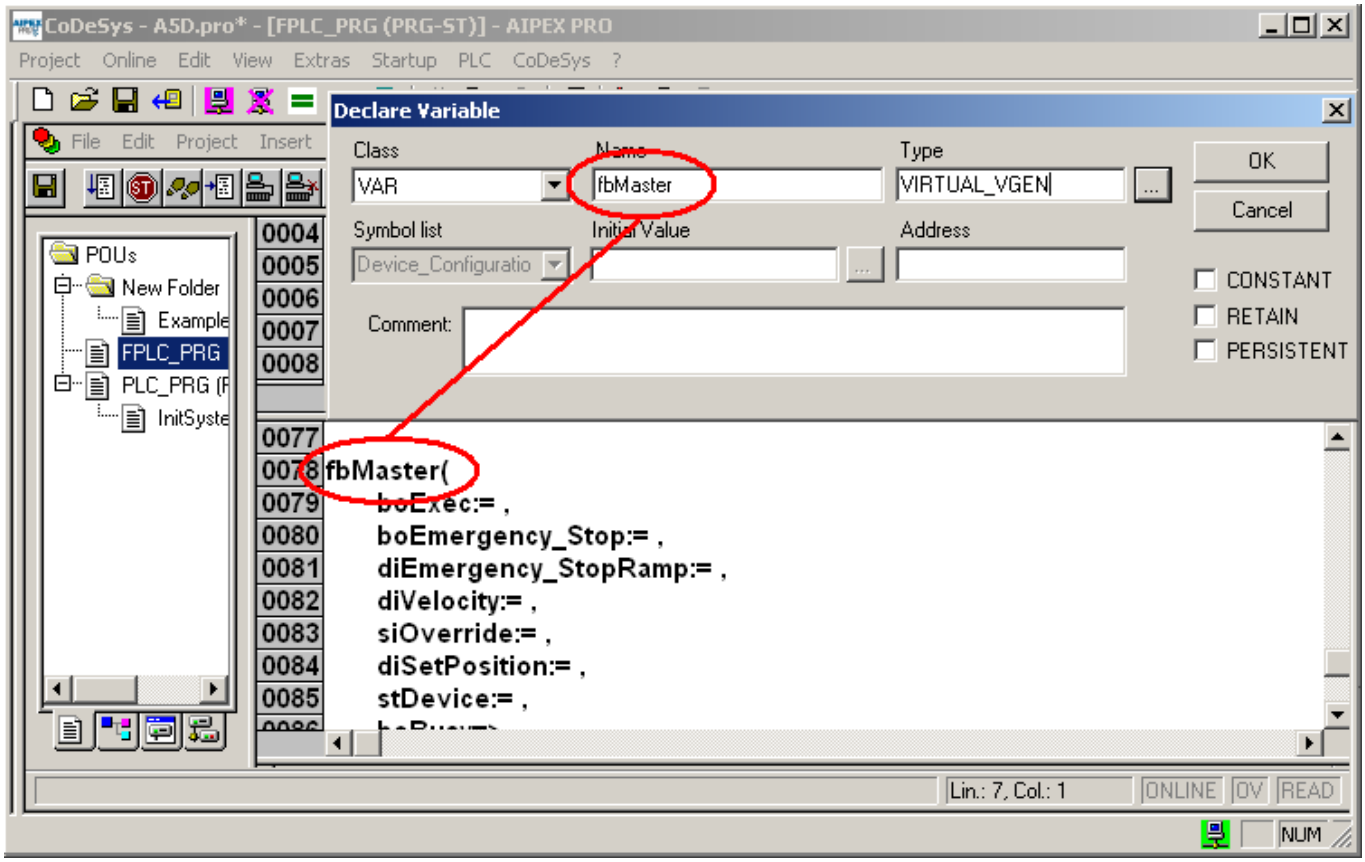
Click with the mouse in the program editor.

Press the **'F2'** button to open the 'Input assistant'.

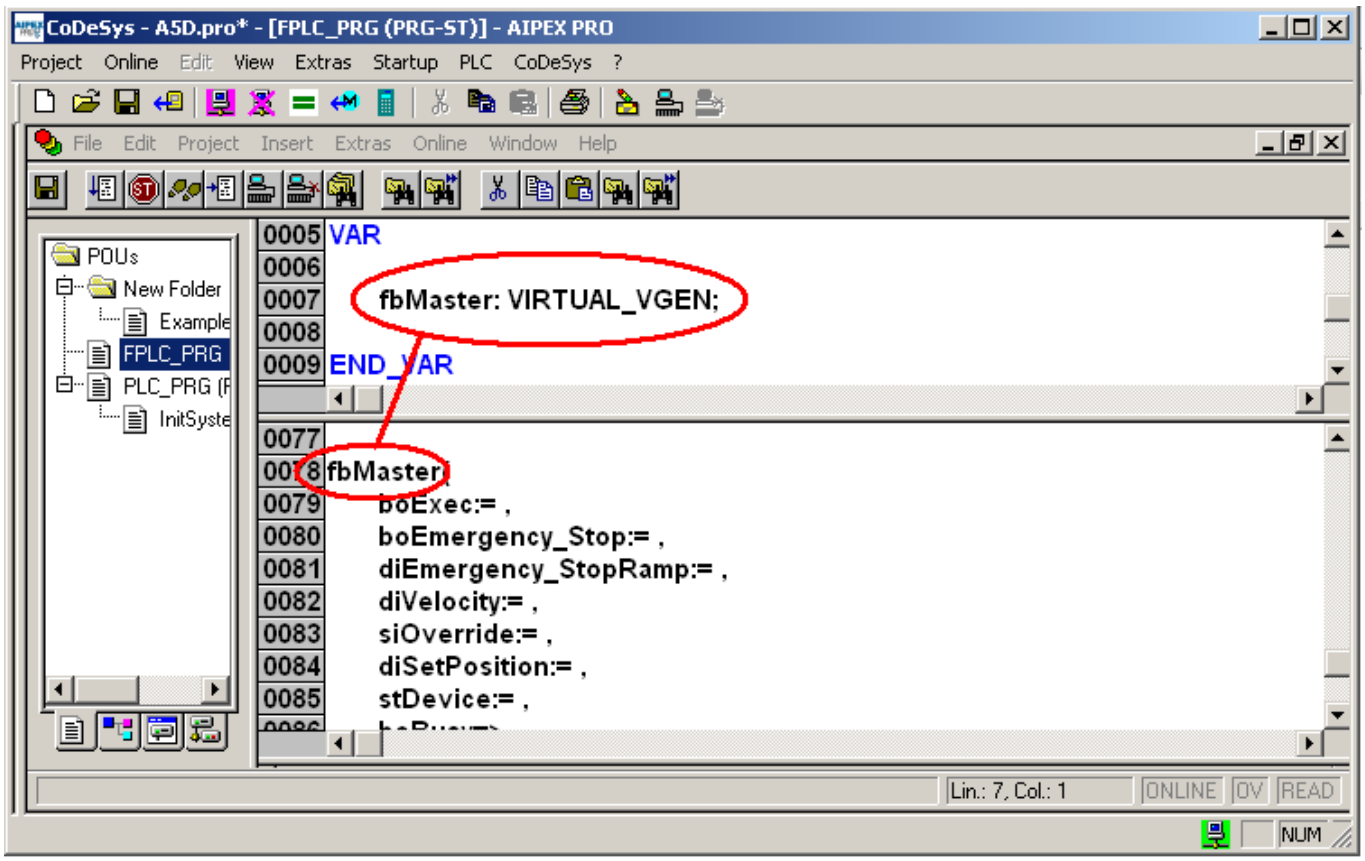
In the standard function blocks, you can find the 'FB VIRTUAL_VGEN' used in the example.



Assign a name to the copy. The original name is entered as type in the 'Declare Variable'.



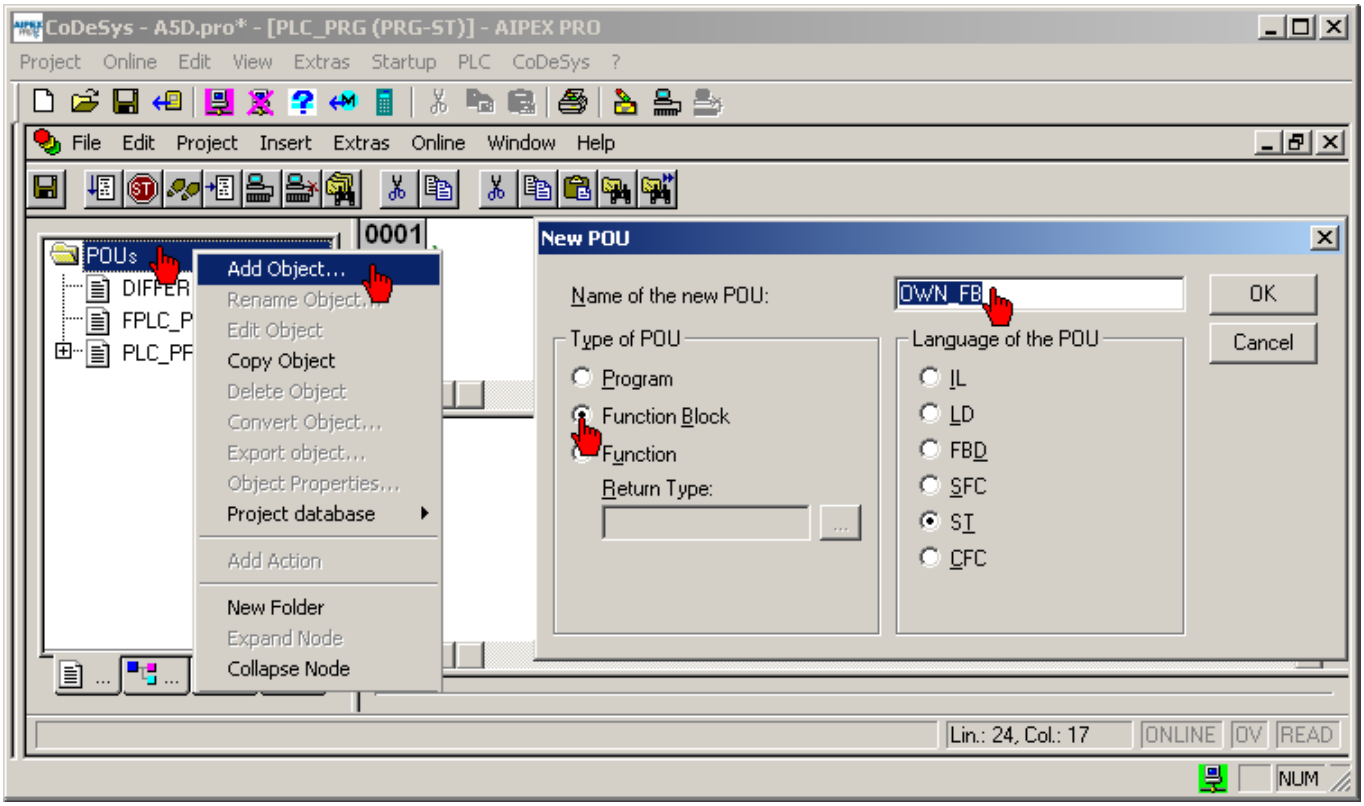

The declaration is entered automatically in the 'Declare Variable window'.



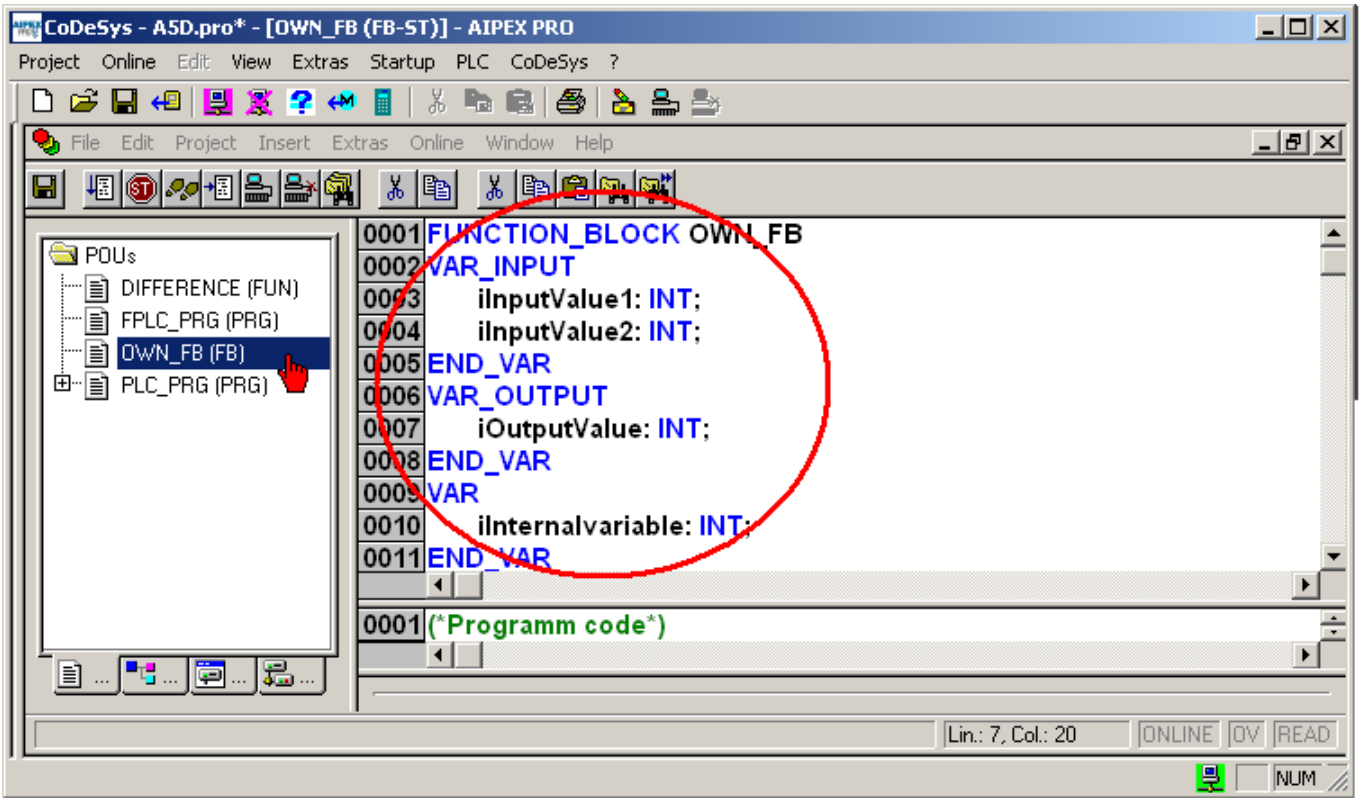
### 4.2.2 User-defined function block

The example shows how you insert a user-defined function block and then invoke it in the 'PLC_PRG'.

Click with the RMB on the **'POUs'** and afterwards on **'Add Object'** to insert a new function block.
Type of POU: Function Block
Name of the new POU: freely selectable
Language of the POU: freely selectable

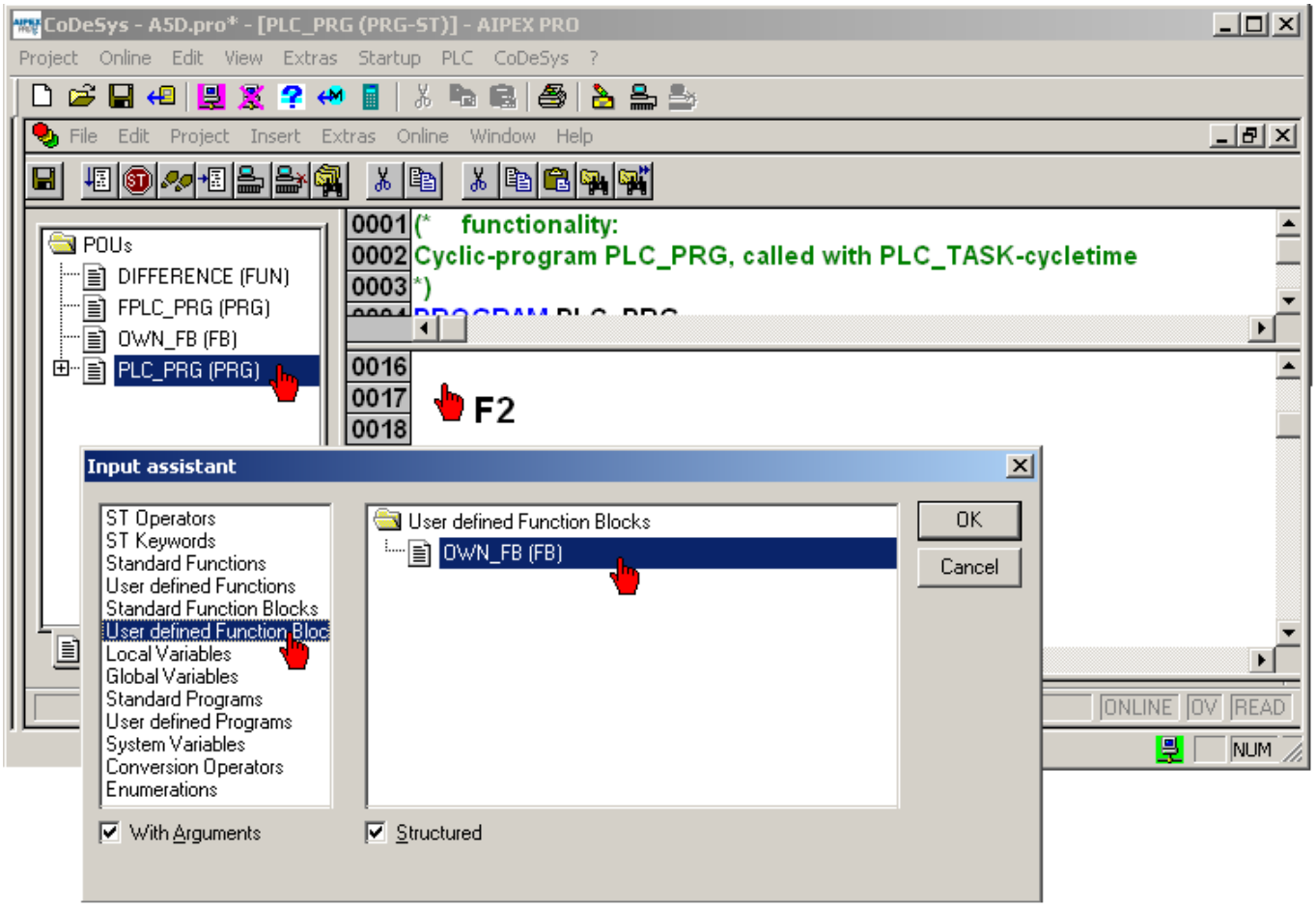


Create the required variable types in the 'Variable declaration window'.
Write your function code in the 'Program editor'.

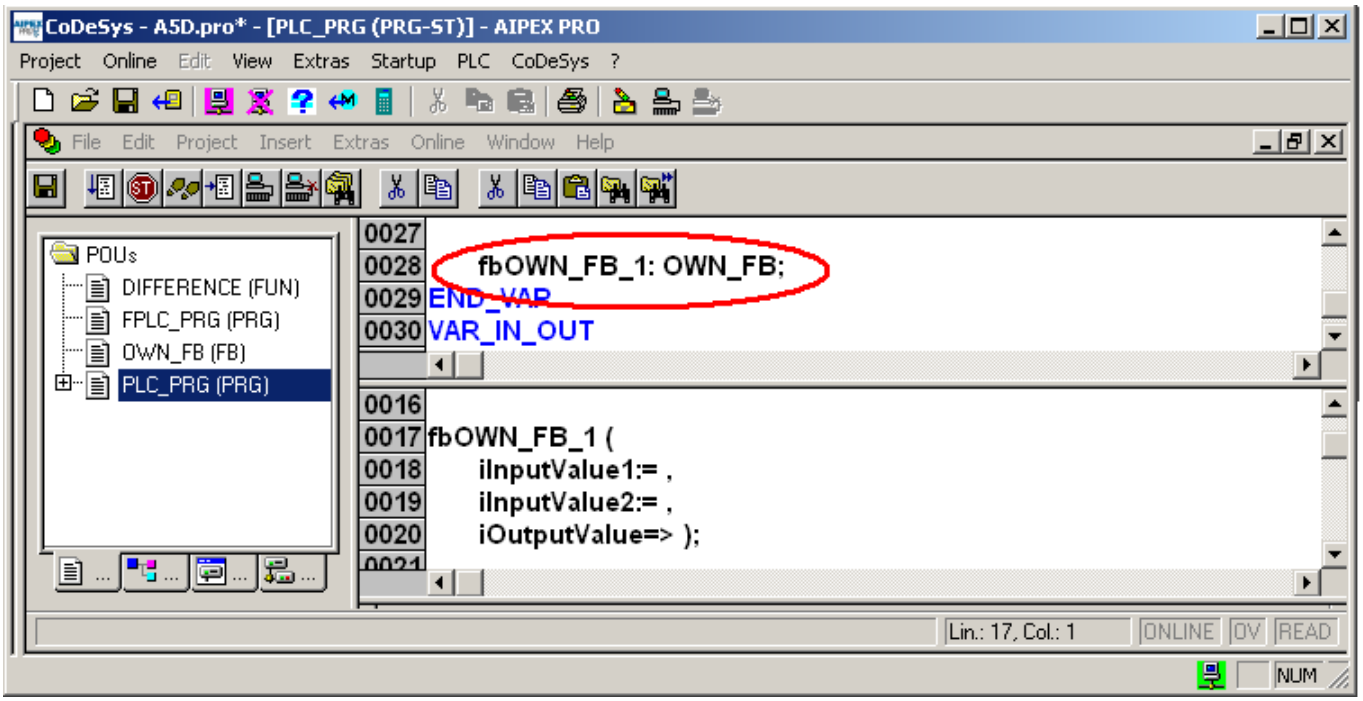



(*Programm code*)
e.g.
iInternalvariable := iInputValue1 + iInputValue2;
iOutputValue := iInternalvariable + 10;

Switch to the **'PLC_PRG'**.
Click with the mouse in the program editor.
Press the **'F2'** button to open the 'Input assistant'.
You will find your FB under the **'User defined Function Blocks'**.



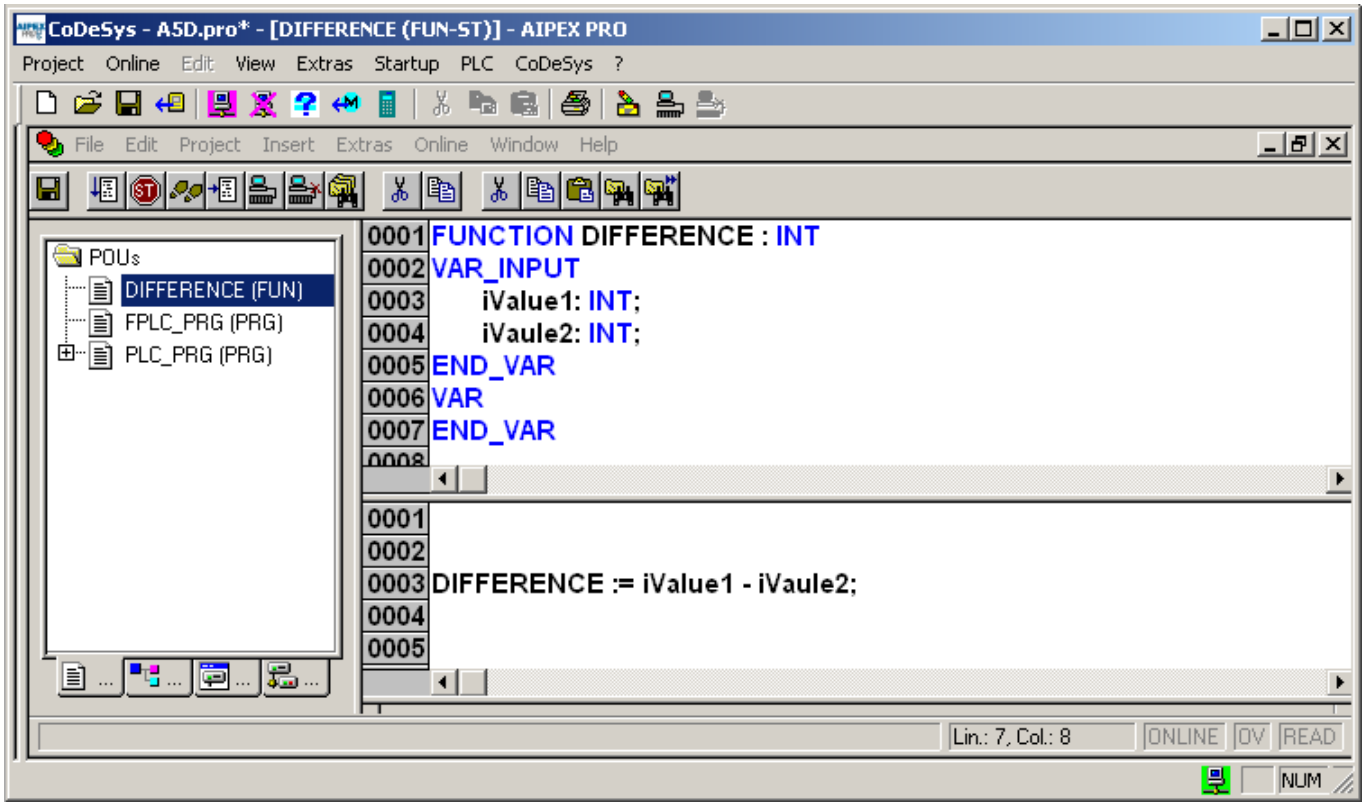

You can instance your FB now as often as you want.
(See Example: Instancing function blocks on page 7.)



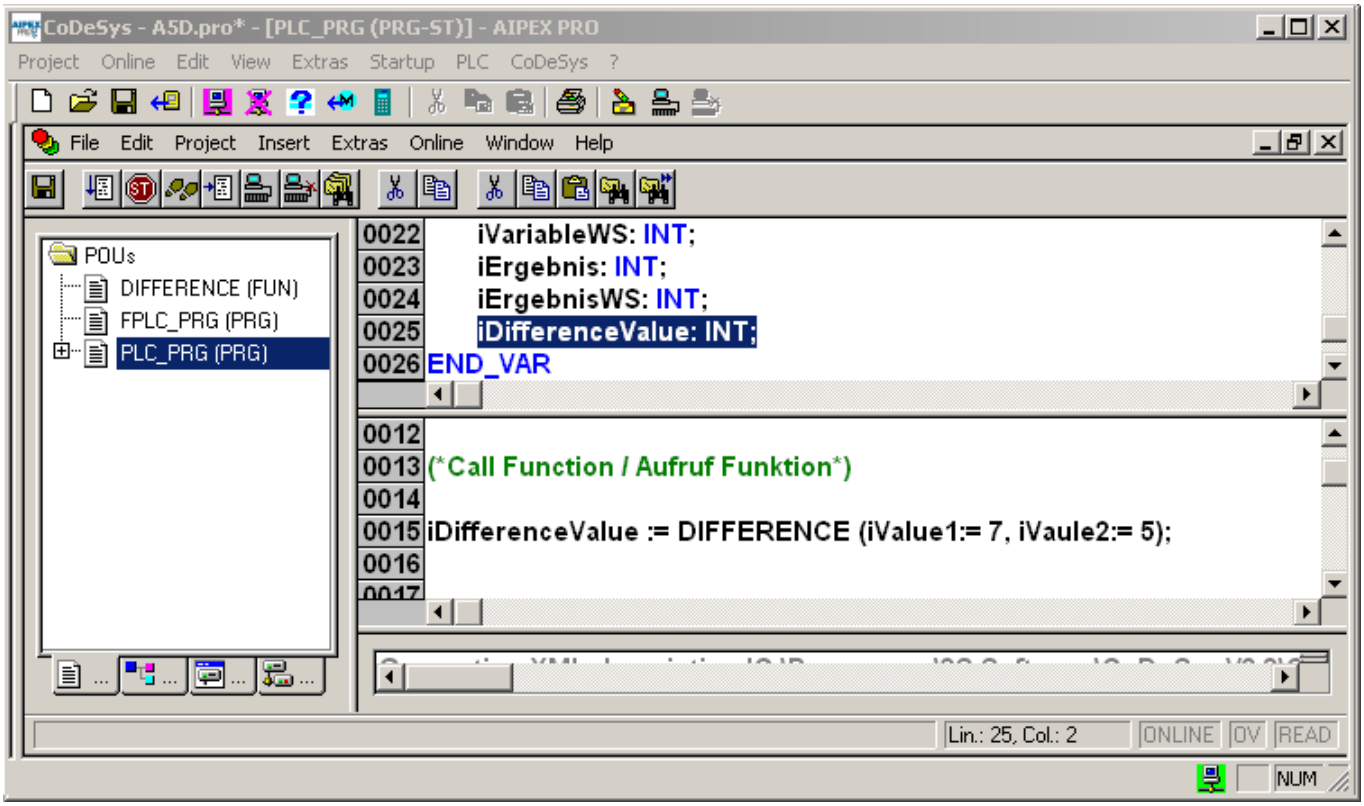
## 4.3 Functions (FUN)

A function is a module that returns exactly one return value as the result of the command.
A function declaration begins with the keyword 'FUNCTION'.
A function is not instanced.

Example:
The user-defined function 'DIFFERENCE' calculates the difference between the input values <iValue1> and <iValue2>.
The return value is an 'INT' type.



Call up the 'DIFFERENCE' function in the 'PLC_PRG'.
Create a variable in which the return value can be transferred.



## 4.4 Actions

Actions can be defined and added to function blocks and programs.

Every action receives a name.

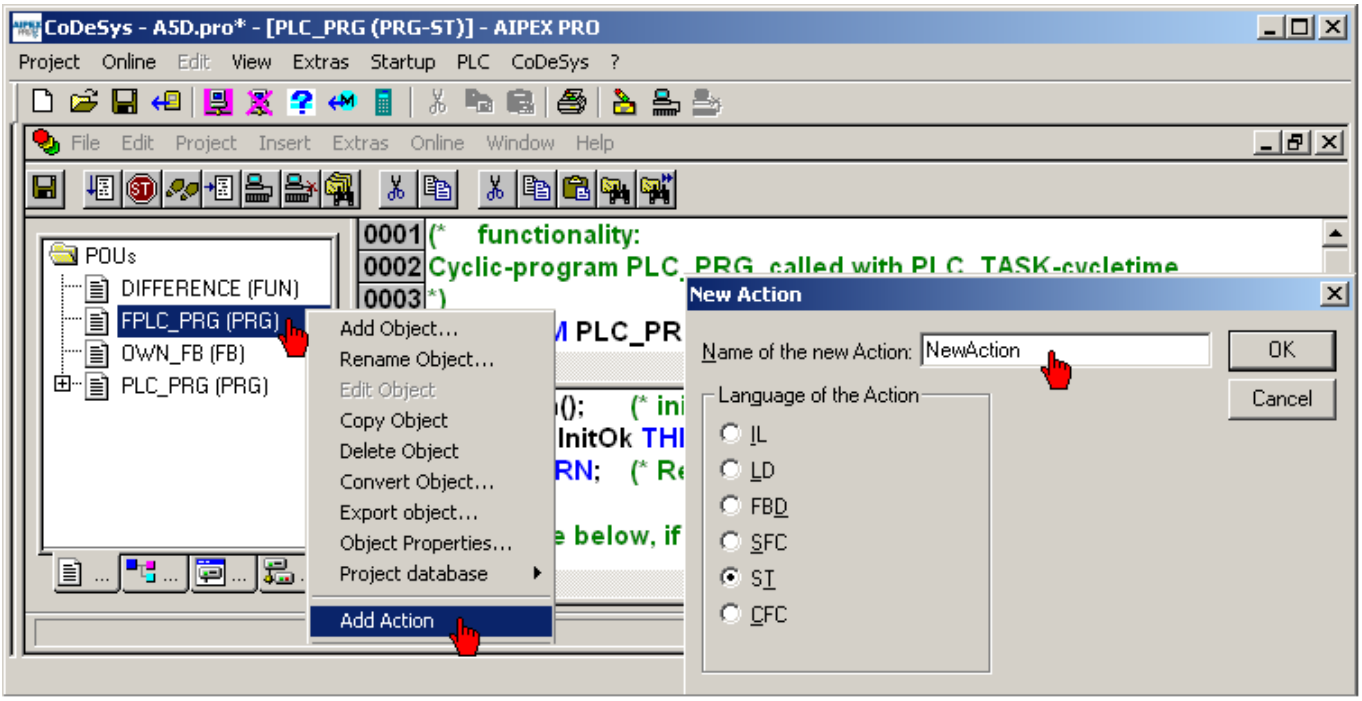
An action operates with the data of the function block or program to which it belongs. The action uses the same input/output variables and local variables as the superordinate module.

Using actions, the program code becomes clearer and more structured. An action needs less program resources. An instancing is not possible.

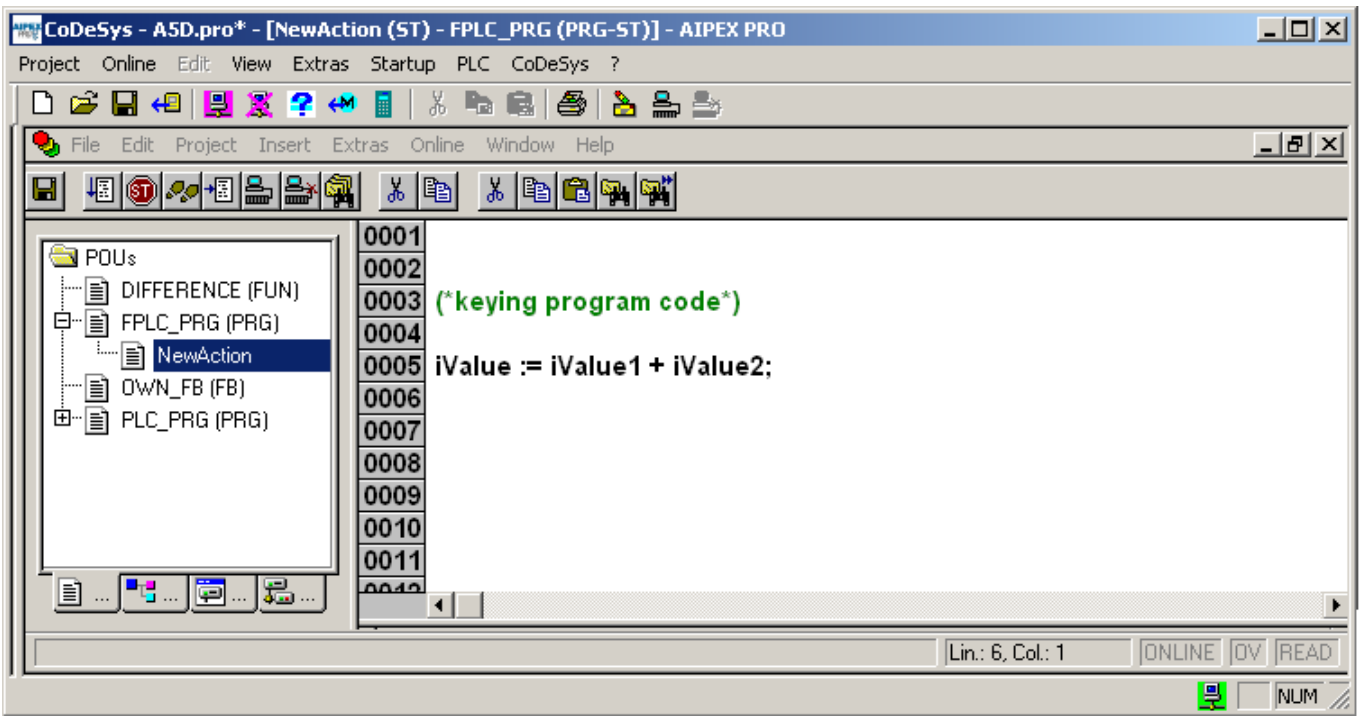
Example: Insert action in the 'FPLC_PRG'.

Click on the **'FPLC_PRG'** with the RMB. Open the 'New Action' window by clicking on **'Add Action'**.

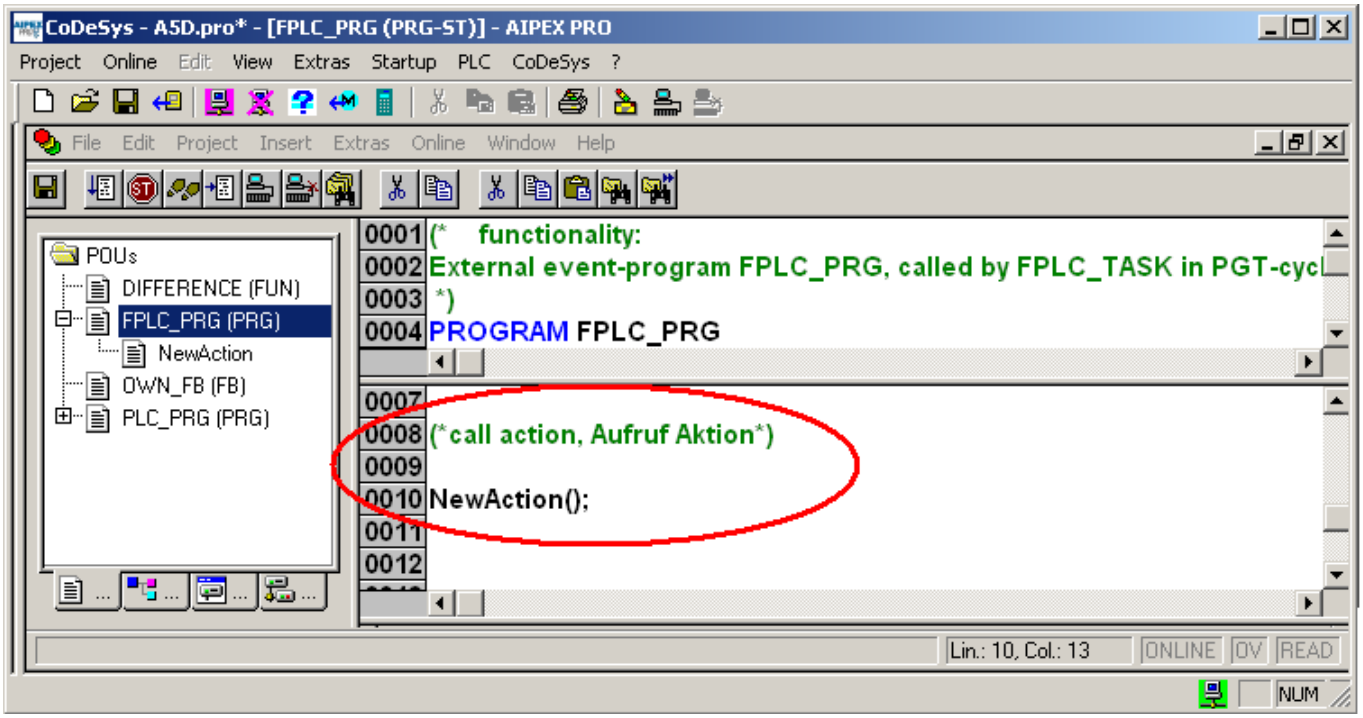
Select the programming language and assign a name then for the new action.




Enter your program code. The declaration of the variables takes place in the superordinate module.



The action is invoked by the name followed by ();. **'NewAction();'**

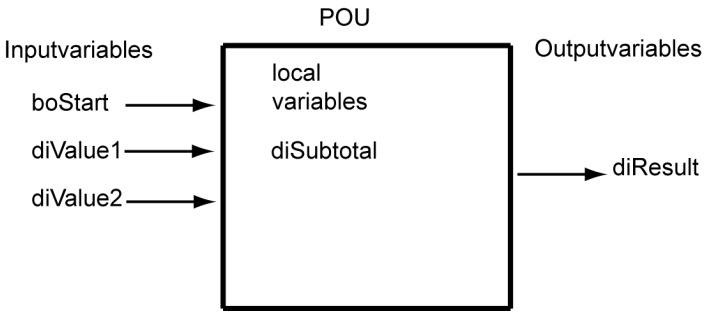


# 5 Variables

## 5.1 Variables

In accordance with IEC 61131-3, each variable that is used in the controller application has to be assigned a class and a type.
Type: See Overview on page 19.

Example module classes:

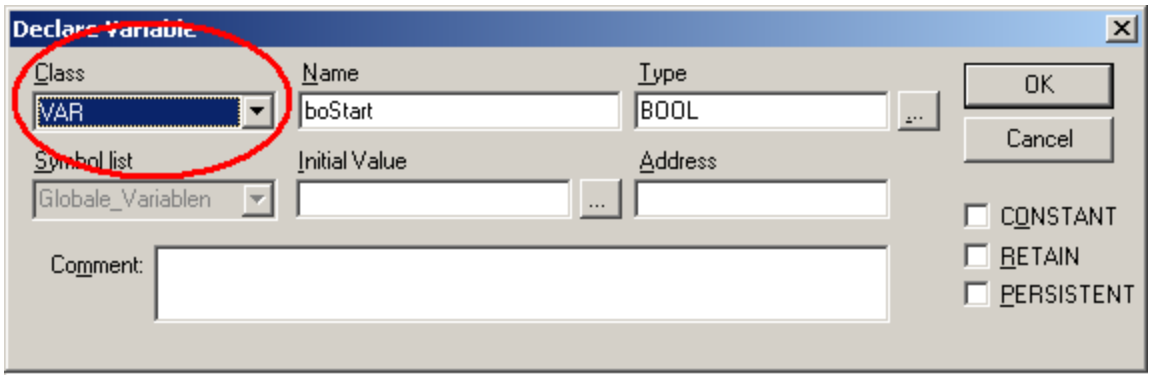



**Local variables**
All local variables of a module are declared between the keywords **'VAR'** and **'END_VAR'**. Local variables have no connection to the outside, i.e., nothing can be written into them from the outside.

**Input variables**
All local variables are declared between the keywords **'VAR_INPUT'** and **'END_VAR'**
that are send along as input variables when the module is called up at the calling point.

**Output variables**
All local variables are declared between the keywords **'VAR_OUTPUT'** and **'END_VAR'**
that serve as output variables of a module, i.e., these values are
returned to the calling module where they can be queried and processed further.

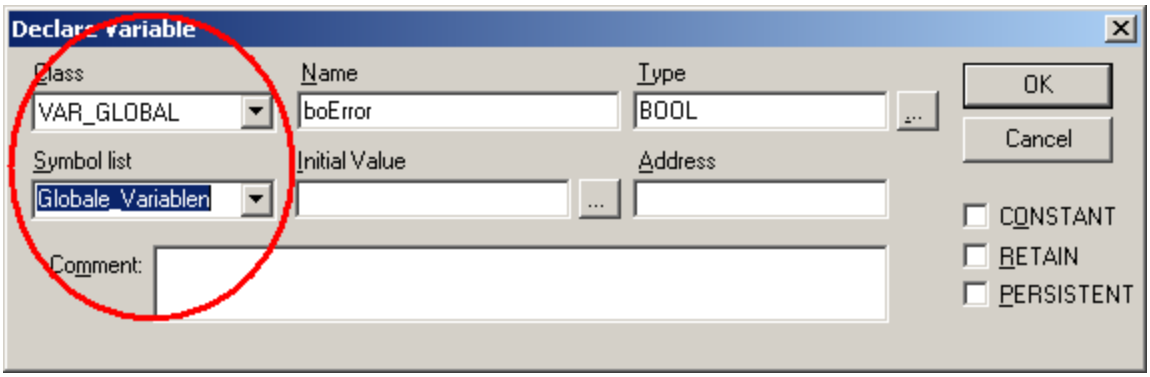
**Input and output variables**
All local variables are declared between the keywords **'VAR_IN_OUT'** and **'END_VAR'**
that serve as input and output variables of a module. For example structures.



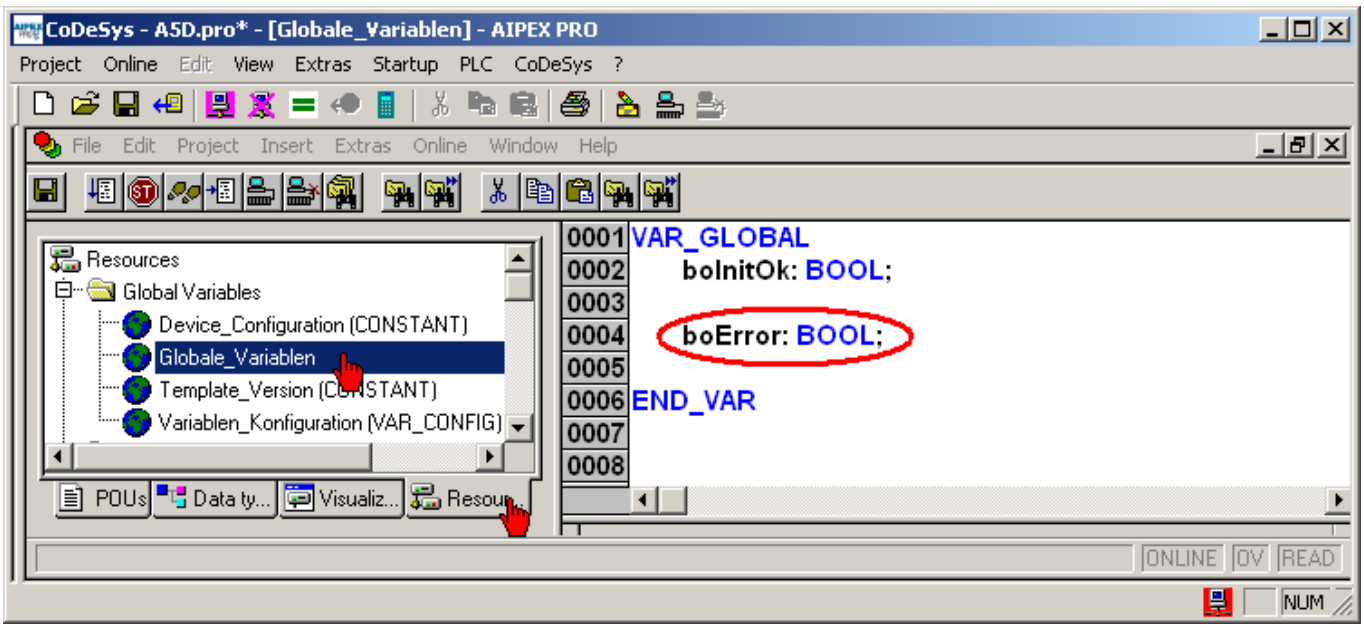
## 5.2 Global variables

"Normal" variables, constants or remanent variables can be declared as global variables that are known in the entire project, but also network variables that additionally serve the data exchange with other network devices.

You can specify the type in the variable declaration window.




The global variables are saved under **'Resources' 'Global Variables'** in the respective directories.



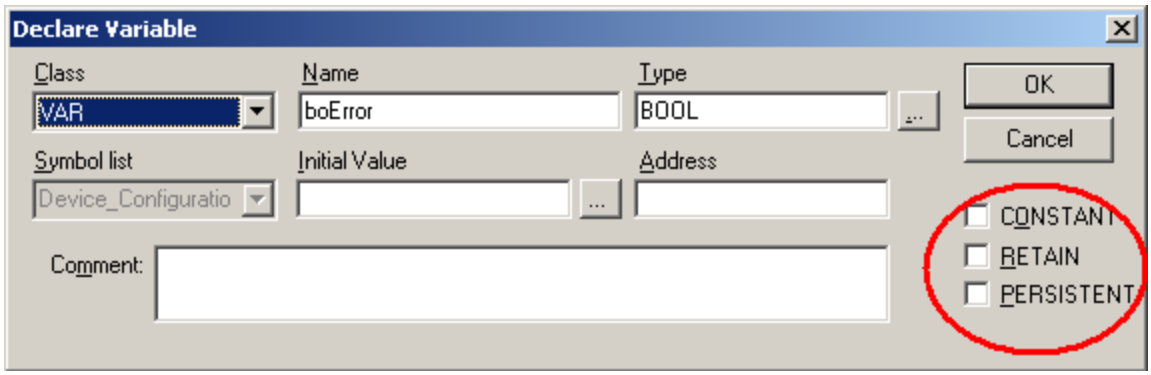
## 5.3 Remanent variables

Remanent variables can keep their value beyond the usual program running time.
This includes retain variables and persistent variables.

| After online command | VAR | VAR RETAIN | VAR PERSISTENT | VAR RETAIN PERSISTENT & VAR PERSISTENT RETAIN |
|---|---|---|---|---|
| Reset<br>Power failure during switching-on/-off | - | x | - | x |
| Reset cold | - | - | - | - |
| Reset source<br>Program is deleted | - | - | - | - |
| Load (=download) | - | - | x | x |
| Online change | x | x | x | x |

x = Value is retained  - = Value is re-initialised

You can specify the type in the variable declaration window.



# 6 Data types

## 6.1 Overview

The naming of variables in all applications and libraries should, if possible, be based on the Hungarian notation:
For each variable, a sensible succinct English description should be found, the basic name. The respective first letter of a word of a basic name should be capitalised; the others written small (example: FileSize). If needed, a translation file can be created additionally in other languages.

In front of these basic names, prefixes are added in small letters depending on the variable's data type.

| Data type | Lower limit | Upper limit | Information content | Prefix | Note |
|---|---|---|---|---|---|
| BOOL | FALSE | TRUE | 1 bit | bo | |
| BYTE | | | 8 bit | by | Bitstring, not for arithm. operations |
| WORD | | | 16 bit | w | Bitstring, not for arithm. operations |
| DWORD | | | 32 bit | dw | Bitstring, not for arithm. operations |
| LWORD | | | 64 bit | lw | Bitstring, not for arithm. operations |
| | | | | | |
| SINT | -128 | 127 | 8 bit | si | |
| USINT | 0 | 255 | 8 bit | usi | |
| INT | -32.768 | 32.767 | 16 bit | i | |
| UINT | 0 | 65.535 | 16 bit | ui | |
| DINT | -2.147.483.648 | 2.147.483.647 | 32 bit | di | |
| UDINT | 0 | 4.294.967.295 | 32 bit | udi | |
| LINT | -263 | 263 - 1 | 64 bit | li | |
| ULINT | 0 | 264 - 1 | 64 bit | uli | |
| | | | | | |
| REAL | | | 32 bit | r | Floating point number |
| LREAL | | | 64 bit | lr | Floating point number |
| | | | | | |
| STRING | | | | s | |
| | | | | | |
| TIME | | | | tim | |
| TIME_OF_DAY | | | | tod | |
| DATETIME | | | | dt | |
| DATE | | | | date | |
| ENUM | | | 16 bit | en | |
| POINTER | | | | p | |
| ARRAY | | | | a | |

## 6.2 Enumeration type

All values of the enumeration type are defined with name when the type is declared (see example: Declaration). An order is specialised thereby (see example: release = 0, config_active = 1 ....) that specifies an order of the individual values.

An enumeration type is known globally in the project.

Example: Declaration

```
VAR
        enState:        (*Enumeration type*)
                (
                Release,                        (*Value 1*)
                Config_active,                  (*Value 2*)
                Config_ended,                   (*Value 3*)
                Main_program_active             (*Value 4*)
                );
END_VAR
```

Example: Program code with CASE loop and enummeration.

```
CASE enState OF

        Release:
                Program code....
                enState := Config_active;
        Config_active:
                Program code....
                enState := Config_ended;
        Config_ended:
                Program code....
                enState := Main_program_active;
        Main_program_active:
                Program code....

END_CASE
```

Advantage: The variable enState accepts the name of the status.

## 6.3 Data field (ARRAY)

An ARRAY (also called data field) serves to save greater amounts of data of the same data type. An ARRAY can contain any number of elements. You need the name of the array and the index to access a single element.

One-, two- and three-dimensional arrays of elementary data types are supported.

Arrays can be defined in the declaration part of a module and in the global variable lists.

By switching arrays ( ARRAY[0..2] OF ARRAY[0..3] OF … ), a maximum of 9 dimensions may be created.

Example:

The array consists of 10 elements, which each may save one INT value.

```
arDiagnosis:            ARRAY [1.. 10] OF INT;
```

Access to arrays:

<ARRAY_Name>.<[element number]>

Example:

arDiagnosis[1];

Example:
The array consists of 10 elements, which each may save the content of the structure ST_AXIS_ERROR.

```
arstDiagnosis:          ARRAY [1.. 10] OF ST_AXIS_ERROR;

TYPE ST_AXIS_ERROR:
STRUCT
        boErr:          BOOL;       (* Error *)
        iAdr:           INT;        (* Address * )
        iDiagNo:        INT;        (* Diagnostics number *)
        diInfo1:        DINT;       (* Additional info 1 *)
END_STRUCT
END_TYPE
```

Access to ARRAYS:
<ARRAY_Name>.<[element number]>.<Structure_Name>.<Component name>
Example:
arDiagnosis[1].ST_AXIS_ERROR.boErr

## 6.4 Structure (STRUCT)

Variables that, for example, belong together are combined to an own data type in a structure.
A structure is known globally in the project.

Structures are deposited as objects (data types) in the Object Organizer under the Data type tab. They begin with the keywords TYPE and STRUCT and end with END_STRUCT and END_TYPE.

Example:

```
TYPE ST_ACTUAL_VALUES :
STRUCT
        diVelocity_feedback_value       DINT;       (* Actual speed *)
        diPosition_feedback_value:      DINT;       (* Actual position *)
        diTorque_feedb_value:           DINT;       (* Actual torque *)
END_STRUCT
END_TYPE
```

Access to structures:
<Structure_Name>.<Component name>
Example:
ST_ACTUAL_VALUES.diVelocity_feedback_value

# 7 Structured text

## 7.1 Structured text (ST)

The structured text consists of a series of instructions that can be executed as in the higher languages according to conditions (IF..THEN..ELSE) or in loops (WHILE..DO).

Example:

```
IF value < 7 THEN
    WHILE value < 8 DO
        value := value + 1;
    END_WHILE;
END_IF;
```

### 7.1.1 Expressions

An expression is a construction that returns a value after its evaluation.
Expressions are a combination of operators and operands. An operand can be a constant, a variable, a function call or a further expression.

Evaluation of expressions
The evaluation of an evaluation is done by processing the operators according to certain binding rules. The operator with the strongest binding is processed first, then the operator with the next weakest binding, and so forth until all operators have been processed.
Operators with equal binding strengths are processed from left to right.

## 7.2 ST operators (overview)

In the following, you will find a table of the ST operators in the sequence of their binding strength:

| Operation | Symbol | Binding strength |
|---|---|---|
| Bracket in | (expression) | Strongest binding |
| Function call | Function name (parameter list) | |
| Exponentiate | EXPT | |
| Negate<br>Complementation | -<br>NOT | |
| Multiply<br>Divide<br>Modulo | *<br>/<br>MOD | |
| Add<br>Subtract | +<br>- | |
| Compare | <,>,<=,>= | |
| Equals<br>Unequal | =<br><> | |
| Bool AND | AND | |
| Bool XOR | XOR | |
| Bool OR | OR | Weakest binding |

## 7.3 ST instructions (overview)

The following table presents in tabular order the instructions possible in ST with one example each:

| Instruction type | Example |
|---|---|
| Assignment | A:= B; CV := CV + 1; C:=SIN(X); |
| Calling up a function block and using the FB output | CMD_TMR(IN := %IX5, PT := 300);<br>A:=CMD_TMR.Q |
| RETURN | RETURN; |
| IF | D:=B*B;<br>IF D<0.0 THEN<br>C:=A;<br>ELSIF D=0.0 THEN<br>C:=B;<br>ELSE C:=D;<br>END_IF; |
| CASE | CASE INT1 OF<br>1: BOOL1 := TRUE;<br>2: BOOL2 := TRUE;<br>ELSE<br>BOOL1 := FALSE;<br>BOOL2 := FALSE;<br>END_CASE; |
| FOR | J:=101;<br>FOR I:=1 TO 100 BY 2 DO<br>IF ARR[I] = 70 THEN<br>J:=I; EXIT;<br>END_IF;<br>END_FOR; |
| WHILE | J:=1;<br>WHILE J<= 100 AND ARR[J] <> 70 DO<br>J:=J+2;<br>END_WHILE; |
| REPEAT | J:=-1;<br>REPEAT<br>J:=J+2;<br>UNTIL J= 101 OR ARR[J] = 70<br>END_REPEAT; |
| EXIT | EXIT; |
| Empty instruction | ; |

## 7.4 ST code

### 7.4.1 Assignment operator :=

On the left side of an assignment, there is an operator (variable, address).
The value of the expression on the right side is assigned to the operator.

Example:
After executing this line, <iVar1> has ten times the value of <iVar2>.

iVar1 **:=** iVar2 * 10;

### 7.4.2 Calling up function blocks in ST

A function block is called up in ST by entering the instance name of the original function block in the programming editor. In the subsequent brackets, you can assign values or variables to the input and output variables. See Example: Instancing function blocks on page 7.

In the example, an FB is called up for parameter reading. It reads the ID116 'Resolution motor encoder' of the <g_stAchse1> and writes the result into the variable <diEncoderresolution>.

```
fbREAD_ID_DINT(
        boExec:= ,
        uiIDNo:= ,          116,
        uiParInst:=         0,
        stDevice:=          g_stAchse1,
        boDone=> ,
        boErr=> ,
        iErrID=> ,
        diIDVal=>           diEncoderresolution);
```

Alternative access via the point operator to the individual elements.

Example:
fbREAD_ID_DINT.boExec := TRUE;
diEncoderresolution := fbREAD_ID_DINT.diIDVal;

FB instancing: See Example: Instancing function blocks on page 7.

### 7.4.3 CASE instruction

Using the CASE instruction, several conditional instructions with the same condition variables can be combined in one construction.

A CASE instruction is processed according to the following scheme:

- If the <iStatus> variable has the value 1, the instruction [1] is processed, value 2 corresponds to instruction [2] etc.
- If the same instruction [3] should be executed for several values of the variable [3,4], then you can write these values after each other separated by commas.
- If the same instruction [4] should be executed for a value range of the variables [5..10], then you can write the start and end values after each other separated by two periods.
- If <iStatus> has none of the specified values, then the ELSE instruction [5] is executed.

```
CASE <iStatus> OF
        1:      Instruction 1
        2:      Instruction 2
        3, 4:   Instruction 3
        5..10:  Instruction 4
ELSE            Instruction 5

END_CASE
```

Example: CASE instruction with integrated IF instructions

```
CASE iStatus OF
        0:      (*Instructions system initialisation *)
                IF boSystemReady THEN
                        iStatus := 1;
                END_IF

        1:      (* Program distribution *)
                IF boSetupMode THEN
                        iStatus := 2;
                        ELSIF boMainProgram THEN
                                iStatus := 3;
                        ELSIF boRestart THEN
                                iStatus := 0;
                END_IF

        2:      (* Instructions Setup mode *)
                IF boMainProgram THEN
                        iStatus := 3;
                        ELSIF boRestart THEN
                                iStatus := 0;
                END_IF

        3:      (* Instructions main program *)
                IF boNeuStart OR boError THEN
                        iStatus := 0;
                END_IF
END_CASE
```

## 7.4.4 IF instruction

With the IF instruction, a condition for 'true' or 'false' can be checked. Instructions can be executed depending on the condition. The part in the curly brackets {} is optional.

- If <Boolean_expression1> returns TRUE, then only the IF instruction is executed and none of the further instructions.
- Otherwise, the Boolean expressions, beginning with <Boolean_expression2>, are evaluated one after the other until one of the expressions results in TRUE. Then only the instructions after this Boolean expression and before the next ELSE or ELSIF is evaluated.
- If none of the Boolean expressions results in TRUE, then the ELSE instruction is executed exclusively.

```
IF <Boolean_expression1> THEN
                IF instruction

{       ELSIF <Boolean_expression2> THEN
                ELSIF instruction 1

        ELSIF <Boolean_expression3> THEN
                ELSIF instruction 2
```

**ELSE** ELSE instruction **}**

**END_IF**

Examples: Boolean expressions

<boStart> = TRUE alternative query <boStart>
<boError> = FALSE alternative query **NOT** <boError>
<boValue> **<** 17

## 7.4.5 FOR loop

Using the FOR loop, you can program repetitive processes.

INT_Var :INT;

**FOR** <INT_Var> := <INIT_WERT>
**TO** <END_VALUE>
{**BY** <increment value>}

**DO** <Instructions>

**END_FOR;**

The part in the curly brackets {} is optional.

- The <instructions> are executed until the counter <INT_Var> reaches the <END_VALUE>.
- This is checked before executing the <instructions> so that the <instructions> are never executed if <INIT_VALUE> is greater than <END_VALUE>.
- Every time <instructions> were carried out, <INT_Var> is increased by <increment value>.
- The increment value can have any integer value. If no value is specified, the increment value is set to the value 1. The loop is ended, because <INT_Var> only becomes greater.

**FOR** iCounterF := 1
**TO** 5
**BY** 1

**DO** iVariableF := iVariableF * 2;

**END_FOR;**

iResultF := iVariableF

Start value: iVariableF = 1;
iResultF = 32;

The <END_VALUE> may not be the limit value of the counter <INT_VAR>. For example, if the variable Counter is of the type SINT, the <END_VALUE> may not be 127, because otherwise there is an infinite loop.

### 7.4.6 WHILE loop

The WHILE loop can be applied like the FOR loop with the difference that the termination condition can be any Boolean expression.

The <instructions> are repeatedly exectuted as long as the <Boolean_expression> results in TRUE.
If the <Boolean_expression> returns FALSE already at the first evaluation, then the <instructions> are never executed.
If the <Boolean_expression> never results in the value FALSE, then the <instructions> are repeated endlessly, causeing a runtime error.

The programmer has to ensure that no infinite loop is created by changing the condition in the instruction part of the loop, for example with a counter that counts up or down.

```
WHILE <Boolean expression>          DO
      <Instructions>
END_WHILE;
```

Example:

Start values: iCounterWL := 5;
iVariableWL := 2;

```
WHILE iCounterWL <> 0 DO
      iVariableWL := iVariableWL * 2;
      iCounterWL := iCounterWL - 1;
END_WHILE
iResultWL := iVariableWL
```

End values: iResultWL := 64;

### 7.4.7 REPEAT loop

The REPEAT loop differs from the WHILE loop in that the termination condition is not checked until after executing the loop. This results in the loop being run through at least once, regardless of the result of the termination condition.

- The <instructions> are exectuted until the <Boolean_expression> results in TRUE.
- If <Boolean_expression> returns TRUE already at the first evaluation, then the <instructions> are executed just once.
- If <Boolean_expression> never results in the value TRUE, then the <instructions> are repeated endlessly, causing a runtime error.

The programmer has to ensure that no infinite loop is created by changing the condition in the instruction part of the loop, for example with a counter that counts up or down.

```
REPEAT      <Instructions>
UNTIL       <Boolean_expression>
END_REPEAT
```

```
REPEAT
        iVariableR := iVariableR * 2;
        iCounterR := iCounterR - 1;

UNTIL   iCounterR = 0

END_REPEAT
```

## 7.4.8 RETURN instruction

With the RETURN instruction, you can exit a module.
For example, if a condition is not met.

## 7.4.9 EXIT instruction

EXIT exits the innermost loop in nested FOR, WHILE or REPEAT loops, regardless of the termination condition.

## Your Opinion is Important!

With our documentation we want to offer you the highest quality support in handling the AMK products.
That is why we are now working on optimising our documentation.
Your comments or suggestions are always interesting for us.
We would be grateful if you take a bit time and answer our questions. Please return a copy of this page to us.

*e-mail: dokumentation@amk-antriebe.de*
or
*fax-No.: +49 (0) 70 21 / 50 05-199*

**Thank you for your assistance.**
**Your AMK documentation team**

1. How would you rate the layout of our AMK documentation?

   (1) very good (2) good (3) satisfactory (4) less than satisfactory (5) poor

2. Is the content structured well?

   (1) very good (2) good (3) moderate (4) hardly (5) not

3. How easy is it to understand the documentation?

   (1) very easy (2) easy (3) moderately easy (4) difficult (5) extremely difficult

4. Did you miss any topics in the documentation?

   (1) no (2) yes, which:

5. How would you rate the overall service at AMK?

   (1) very good (2) good (3) satisfactory (4) less than satisfactory (5) poor

AMK Arnold Müller GmbH & Co. KG
phone : +49 (0) 70 21 / 50 05-0, fax: +49 (0) 70 21 / 50 05-199,
info@amk-antriebe.de